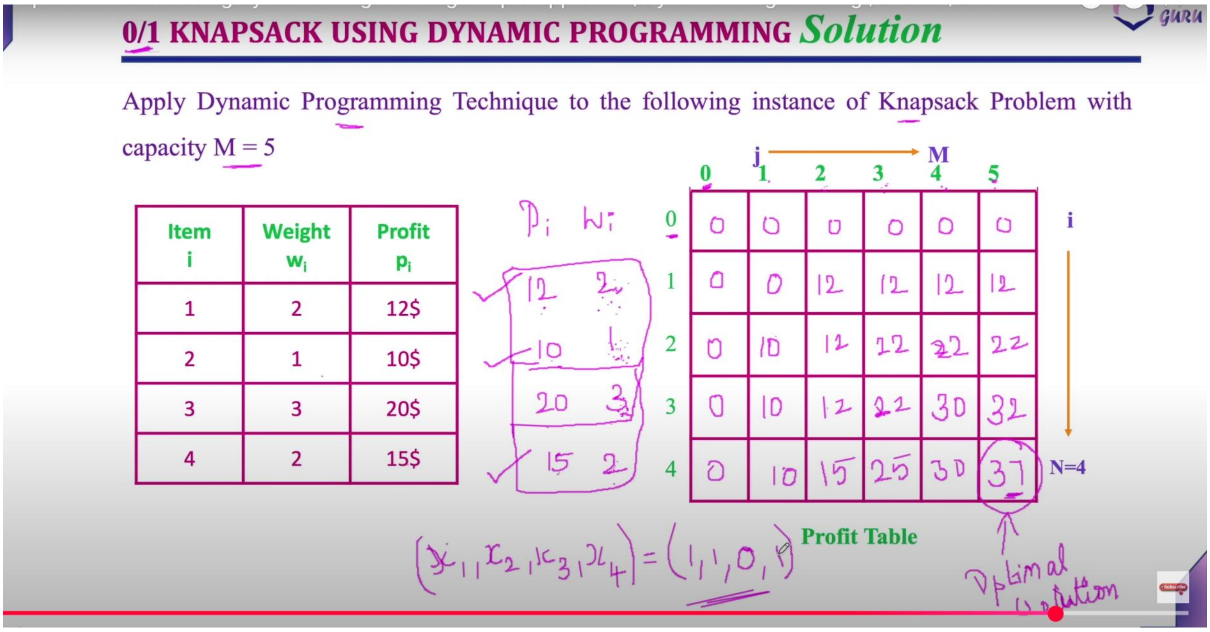
Unit 4

Dynamic programming

01 knapsack



Youtube link :- https://youtu.be/4YaV8eUaM7Q?feature=shared

📌 0/1 Knapsack – Pseudocode (Dynamic Programming)

plaintext Copy Edit

```
function knapsack(weights[1...n], values[1...n], W):
    // n = number of items
    // W = total capacity of knapsack
    // weights[i] = weight of item i
    // values[i] = value of item i

    Create a 2D array dp[0...n][0...W] initialized to 0

    for i from 1 to n:
        for w from 0 to W:
            if weights[i] ≤ w:
                dp[i][w] = max(dp[i-1][w], values[i] + dp[i-1][w - weights[i]])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][W]  // Maximum value that can be put in knapsack
```

✅ Time Complexity of 0/1 Knapsack (Dynamic Programming)

Let:

- `n` = number of items
- `W` = capacity of the knapsack

Steps & Time Complexity:

1. **Create a DP table of size `[n+1][W+1]`:**
   - Each cell `dp[i][w]` stores the max value for first `i` items and capacity `w`.

   🔹 **Time = O(n × W)**

2. **Fill the DP table using nested loops:**
   - Outer loop runs `n` times (items)
   - Inner loop runs `W` times (capacities)

   🔹 **Time = O(n × W)**

🔚 Total Time Complexity:

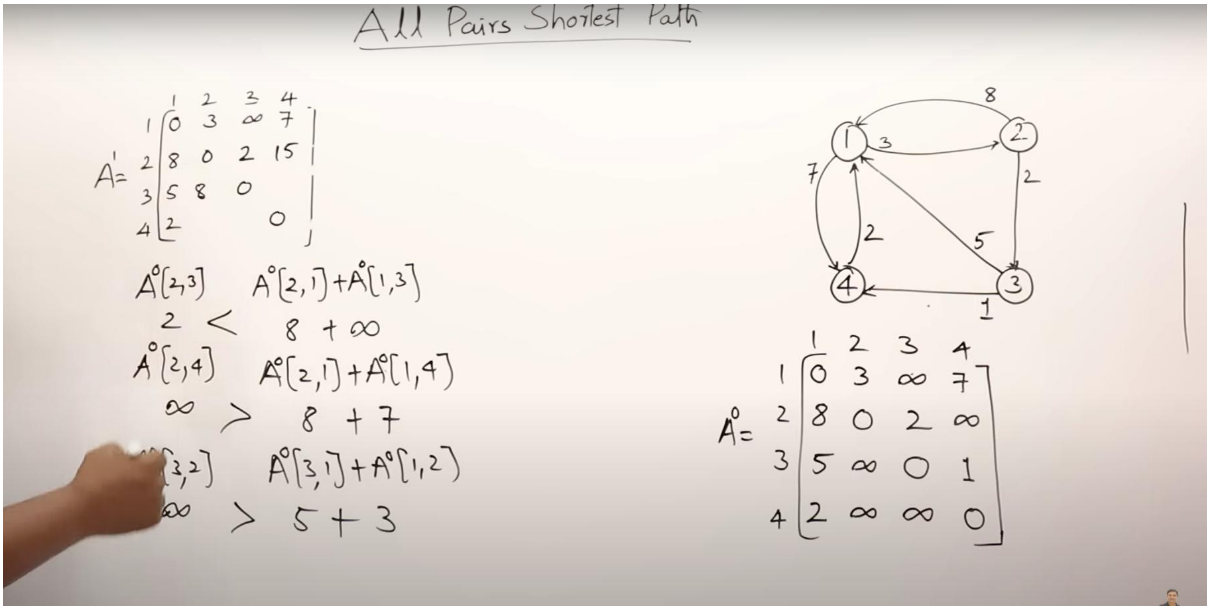
```plaintext
O(n × W)
```

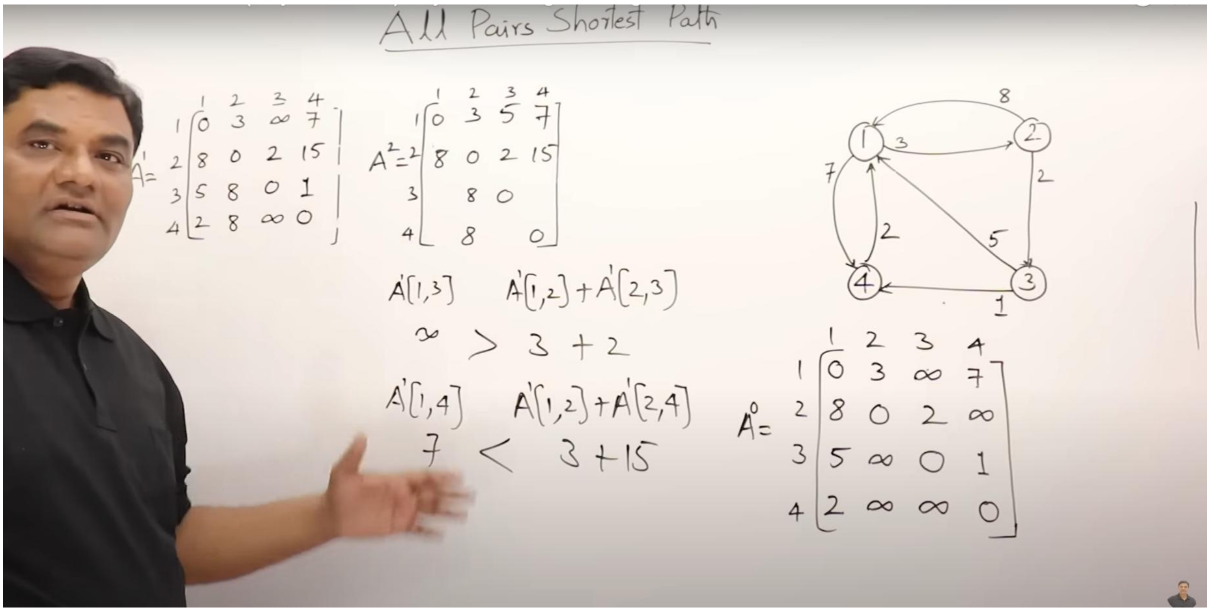
✅ Final Answer:

> **Time Complexity = O(n × W)**

This is **pseudo-polynomial time** since it's linear in `W`, which can be large.

Floyd warshall algorithm





Youtube link :- https://youtu.be/oNI0rf2P9gE?feature=shared plaintext

```
function floydWarshall(graph[0...n-1][0...n-1]):
    // graph[i][j] is the weight of the edge from i to j
    // If no edge exists between i and j, use ∞ (infinity)
    // n = number of vertices

    dist[0...n-1][0...n-1] = graph

    for k from 0 to n - 1:
        for i from 0 to n - 1:
            for j from 0 to n - 1:
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

Let:

- `n` = number of vertices in the graph

Steps & Time Complexity:

1. Three nested loops:

plaintext

```
for k from 0 to n-1:
    for i from 0 to n-1:
        for j from 0 to n-1:
            // Update dist[i][j]
```

- Each loop runs `n` times
  - Time = $O(n^3)$

Total Time Complexity:

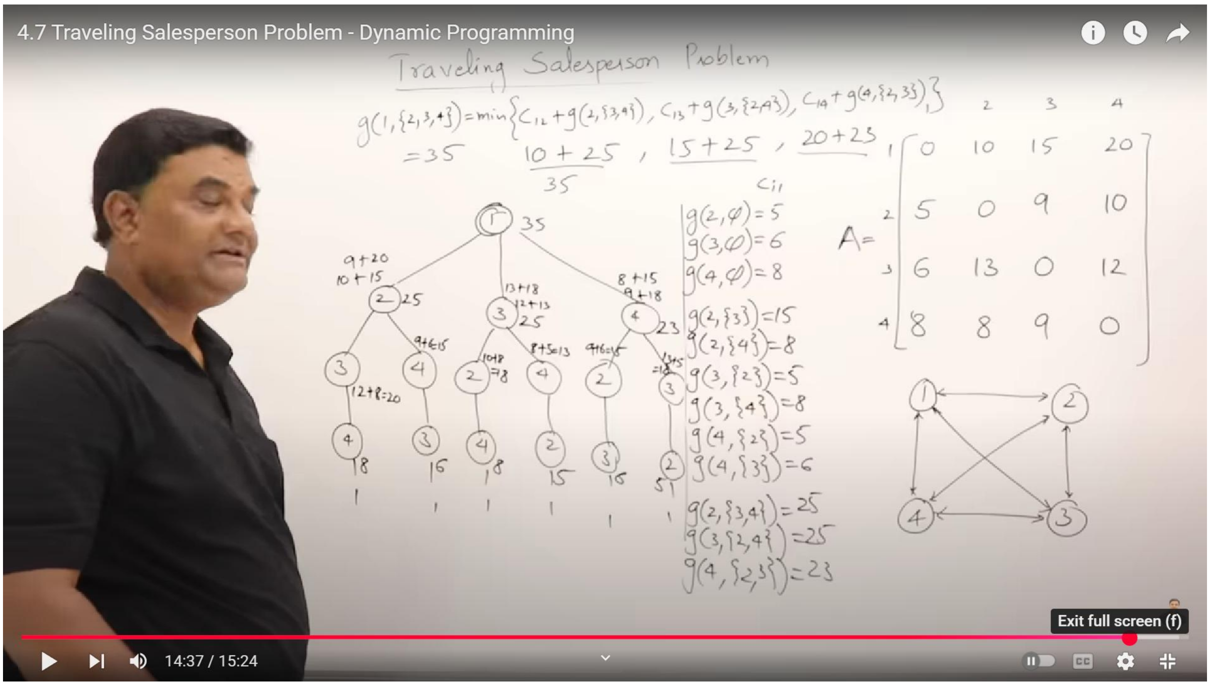
```plaintext
O(n³)
```

✅ Final Answer:

> **Time Complexity = O(n^3)**

This is because the algorithm checks **every pair (i, j)** for **every intermediate node k**.

Travelling Salesman



📌 Travelling Salesman Problem – Pseudocode (Dynamic Programming + Bitmasking)

plaintext

Copy Edit

```
function tsp(dp, dist, visitedMask, currCity, n):
    // dp[mask][city] stores the minimum cost to reach `city`
    // after visiting cities in `mask`

    if visitedMask == (1 << n) - 1:
        return dist[currCity][0]  // return to starting city

    if dp[visitedMask][currCity] ≠ -1:
        return dp[visitedMask][currCity]
```

```
    minCost = ∞
    for nextCity from 0 to n-1:
        if visitedMask does not include nextCity:
            newMask = visitedMask | (1 << nextCity)
            cost = dist[currCity][nextCity] + tsp(dp, dist, newMask, nextCity, n)
            minCost = min(minCost, cost)

    dp[visitedMask][currCity] = minCost
    return minCost
```

✅ Time Complexity of TSP (DP with Bitmasking)

Let:

- `n` = number of cities

How it works:

- There are 2^n subsets (bitmasks) of visited cities
- For each subset, we compute cost for each of the `n` possible last cities
- Each call takes `O(n)` time to try next cities

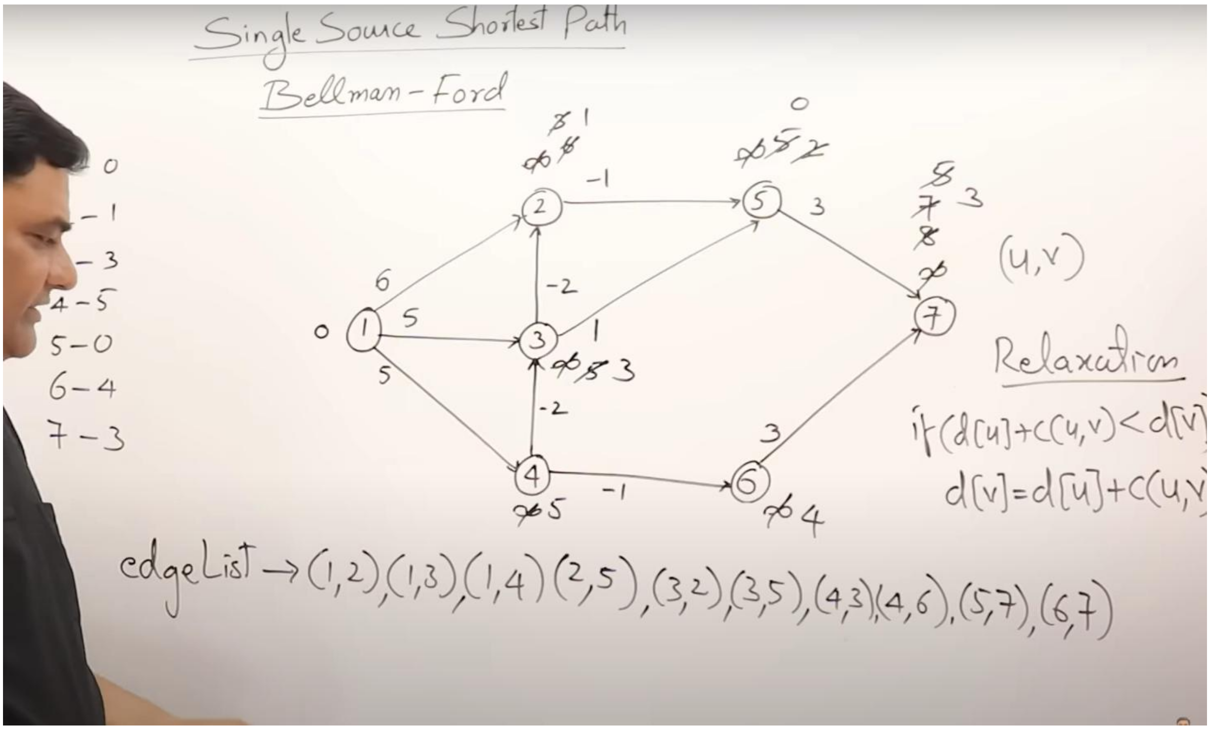
Total Time Complexity:

```plaintext
O(n² × 2ⁿ)
```

Final Answer:

> **Time Complexity = O(n^2 × 2^n)**
> This is the most efficient **exact** approach for TSP (Held-Karp).

Bellman ford algorithm



Youtube link :- https://youtu.be/FtN3BYH2Zes?feature=shared

```
function bellmanFord(graph, V, E, source):
    // graph is a list of E edges: each edge is (u, v, weight)
    // V = number of vertices
    // E = number of edges
    // source = starting vertex

    dist[0...V-1] = ∞
    dist[source] = 0

    // Step 1: Relax all edges V - 1 times
    for i from 1 to V - 1:
        for each edge (u, v, w) in graph:
            if dist[u] ≠ ∞ and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
```

Copy Edit

```
    // Step 2: Check for negative weight cycles
    for each edge (u, v, w) in graph:
        if dist[u] ≠ ∞ and dist[u] + w < dist[v]:
            return "Negative weight cycle detected"

    return dist
```

✅ Time Complexity of Bellman-Ford

Let:

- `V` = number of vertices
- `E` = number of edges

🔸 Step-by-Step Complexity:

1. **Initialize distance array →** `O(V)`
2. **Relax all edges V - 1 times →** `O(V × E)`
3. **Check for negative cycles →** `O(E)`

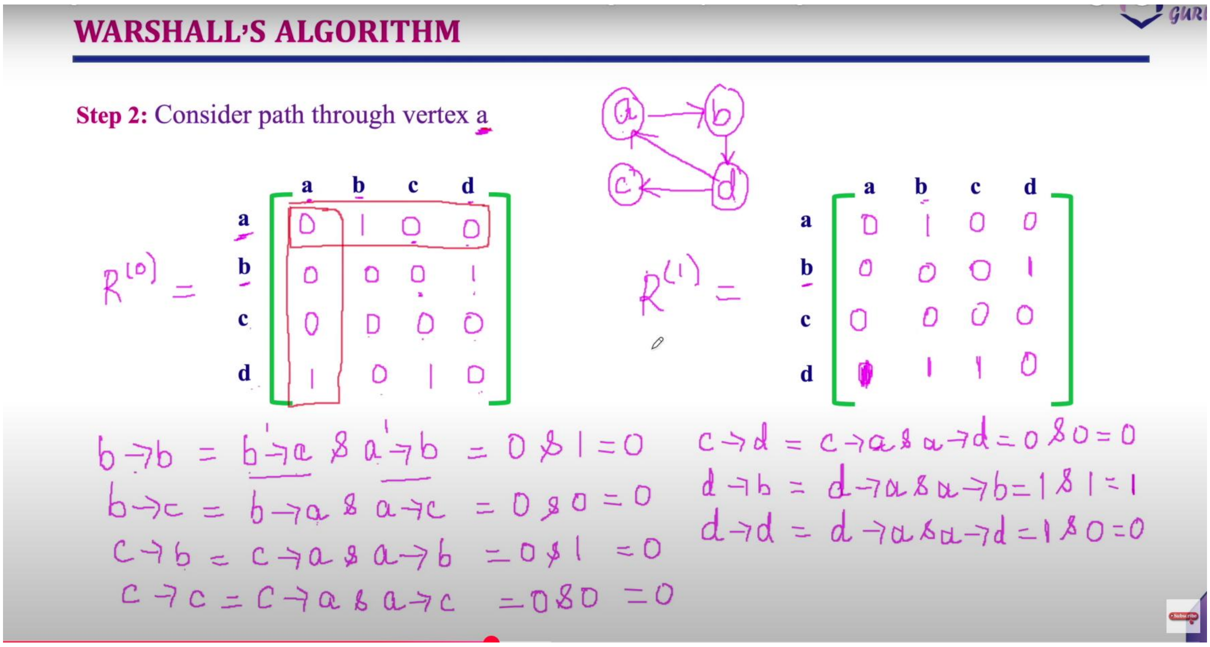
Total Time Complexity:

```plaintext
O(V × E)
```

✅ Final Answer:

> Time Complexity = O(V × E)

Warshall's algorithm



📌 Warshall's Algorithm – Pseudocode

```plaintext
function warshall(graph[0...n-1][0...n-1]):
    // graph[i][j] = 1 if there is a direct edge from i to j, else 0
    // graph will be updated to store the transitive closure

    for k from 0 to n - 1:
        for i from 0 to n - 1:
            for j from 0 to n - 1:
                if graph[i][k] == 1 and graph[k][j] == 1:
                    graph[i][j] = 1  // there is a path from i to j through k

    return graph  // transitive closure matrix
```

Time Complexity of Warshall's Algorithm

Let:

- `n` = number of vertices

Steps:

Three nested loops:

plaintext Copy Edit

```
for k from 0 to n-1:
    for i from 0 to n-1:
        for j from 0 to n-1:
```

Each loop runs `n` times.

Total Time Complexity:

plaintext Copy Edit

```
O(n³)
```

Final Answer:

> Time Complexity = $O(n^3)$

Optimal binary search tree

```
function greedyOBST(keys[1...n], freq[1...n], low, high):
    if low > high:
        return NULL

    // Step 1: Find key with maximum frequency
    maxFreqIndex = low
    for i from low to high:
        if freq[i] > freq[maxFreqIndex]:
            maxFreqIndex = i

    // Step 2: Make this key the root
    root = new Node(keys[maxFreqIndex])

    // Step 3: Recursively build left and right subtrees
    root.left = greedyOBST(keys, freq, low, maxFreqIndex - 1)
    root.right = greedyOBST(keys, freq, maxFreqIndex + 1, high)

    return root
```

✅ Time Complexity of Greedy OBST

- For each recursive call, we find the max frequency in `O(n)` time
- This splits the array into subarrays recursively

🔹 Worst-case: unbalanced calls → like QuickSort

🔹 Time = `O(n²)` in worst case

✅ Final Answer:

> Greedy OBST Time Complexity = $O(n^2)$
> ❌ Does not guarantee optimal cost

Youtube :- https://youtu.be/vLS-zRCHo-Y?feature=shared